UCRL-ID-137578

Scientific Software Component Technology

S. Kohn, D. Gannon, N. Dykman, G. Kumfert and B. Smolinski

February 16, 2000

Work performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

LDRD ERD FY99 Final Report (99-ERD-078)

Scientific Software Component Technology

Principal Investigator: Scott Kohn
Center for Applied Scientific Computing
Computations Directorate

Technical Advisor: Dennis Gannon
Indiana University and NASA Ames Research Center

Co-Investigators: Nathan Dykman, Gary Kumfert and Brent Smolinski
Center for Applied Scientific Computing
Computations Directorate

Funding Summary: FY99: $200K

Executive Summary

We are developing new software component technology for high-performance parallel scientific computing to address issues of complexity, re-use, and interoperability for laboratory software. Component technology enables cross-project code re-use, reduces software development costs, and provides additional simulation capabilities for massively parallel laboratory application codes. The success of our approach will be measured by its impact on DOE mathematical and scientific software efforts. Thus, we are collaborating closely with library developers and application scientists in the Common Component Architecture forum, the Equation Solver Interface forum, and other DOE mathematical software groups to gather requirements, write and adopt a variety of design specifications, and develop demonstration projects to validate our approach.

Numerical simulation is essential to the science mission at the laboratory. However, it is becoming increasingly difficult to manage the complexity of modern simulation software. Computational scientists develop complex, three-dimensional, massively parallel, full-physics simulations that require the integration of diverse software packages written by outside development teams. Currently, the integration of a new software package, such as a new linear solver library, can require several months of effort.

Current industry component technologies such as CORBA, JavaBeans, and COM have all been used successfully in the business domain to reduce software development costs and increate software quality. However, these existing industry component infrastructures will not scale to support massively parallel applications in science and engineering. In particular, they do not address issues related to high-performance parallel computing on ASCI-class machines, such as fast in-process connections between components, language interoperability for scientific languages such as Fortran, parallel data redistribution between components, and massively parallel components. While industrial component systems do not directly address scientific computing issues, we leverage existing industry technologies and design concepts whenever possible.

Since the mid-year start of this project in FY99, we have focused on the needs of seamless language interoperability in a high-performance environment. Computational scientists are routinely hindered in code re-use by differences in programming languages; for example, a solver library written in C++ can be difficult to call from an applications code written in C or Fortran. Our approach adopts the industry practice of using an Interface Definition Language (IDL) to describe component interfaces in a language-independent manner. We have developed an IDL for scientific applications (SIDL) that focuses on the unique needs of the scientific domain as compared to the business world. We have also created tools that use SIDL descriptions of software components to generate automatically language bindings and code that allows the component to be called easily from different languages.

As a demonstration project, we worked with the HYPRE team at LLNL to develop Fortran bindings for their linear solver preconditioning library. Previously, HYPRE did not support a Fortran calling interface, so Fortran programmers did not have access to its advanced solver capabilities. In an afternoon, we were able to define SIDL interfaces for the HYPRE structured grid preconditioning routines and generate the appropriate Fortran calling interface. Performance measurements on the ASCI Blue Pacific platform demonstrate that the overhead of our approach is less than a percent of the overall run-time.

In the following year, we will continue the development of our component interoperability tools and focus on the needs of communication between distributed components running in different processor spaces. We will also provide language support for Python, a popular

scripting language for scientific simulations at the laboratory. Finally, we will continue our collaborations with the Common Component Architecture and Equation Solver Interface forums to help deploy our software component technology across the DOE complex.

Research Papers

We have written refereed two papers describing our activities. These papers are attached to this final report.

The first paper provides an overview of our activities with the Common Component Architecture forum to develop a component architecture for scientific computing. This paper was published in the High Performance Distributed Computing Conference in August of 1999. The release number is UCRL-JC-134475.

Toward a Common Component Architecture for High Performance Scientific Computing, Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois Mcinnes, Steve Parker, and Brent Smolinski, High Performance Distributed Computing Conference, August 1999.

Abstract: This paper describes work in progress to develop a standard for interoperability among high-performance scientific components. This research stems from growing recognition that the scientific community needs to better manage the complexity of multidisciplinary simulations and better address scalable performance issues on parallel and distributed architectures. Driving forces are the need for fast connections among components that perform numerically intensive work and for parallel collective interactions among components that use multiple processes or threads. This paper focuses on the areas we believe are most crucial in this context, namely, an interface definition language that supports scientific abstractions for specifying component interfaces and a ports connection model for specifying component interactions.

The second paper describes our LLNL language interoperability work in some detail. In particular, we demonstrate that our language interoperability approach can work for a scientific library running on an ASCI-class parallel computer. This paper was published in the International Symposium on Object-Oriented Parallel Environments in December of 1999. The release number is UCRL-JC-134260.

Language Interoperability for High-Performance Parallel Scientific Components, Brent Smolinski, Scott Kohn, Noah Elliott, and Nathan Dykman, International Symposium on Object-Oriented Parallel Environments (ISOPE), December 1999.

Abstract: Component technologies offer a promising approach for managing the increasing complexity and interdisciplinary nature of high-performance scientific applications. Language interoperability provides the flexibility required by component architectures. In this paper, we present an approach to language interoperability for high-performance parallel components. Based on Interface Definition Language (IDL) techniques, we have developed a Scientific IDL (SIDL) that focuses on the abstractions and performance requirements of the scientific domain. We have developed a SIDL compiler and the associated run-time support for reference counting, reflection, object management, and basic exception handling. The SIDL approach has been validated for a scientific linear solver library. Initial timing results indicate that the performance overhead is minimal (less than 1%), whereas the savings in development time for interoperable software libraries can be substantial.

Toward a Common Component Architecture for High-Performance Scientific Computing *

Rob Armstrong† Dennis Gannon‡ Al Geist§ Katarzyna Keahey¶ Scott Kohn‖
Lois McInnes** Steve Parker†† Brent Smolinski‡‡

Abstract

This paper describes work in progress to develop a standard for interoperability among high-performance scientific components. This research stems from growing recognition that the scientific community needs to better manage the complexity of multidisciplinary simulations and better address scalable performance issues on parallel and distributed architectures. Driving forces are the need for fast connections among components that perform numerically intensive work and for parallel collective interactions among components that use multiple processes or threads. This paper focuses on the areas we believe are most crucial in this context, namely, an interface definition language that supports scientific abstractions for specifying component interfaces and a ports connection model for specifying component interactions.

1 Introduction

The complexity and resource demands of present-day software systems create the need for more flexible solutions than those offered by conventional programming styles based on a succession of subroutine calls. One solution is component programming, based on encapsulating units of functionality and providing a meta-language specification of their interfaces. Component-based software development can be considered an evolutionary step beyond object-oriented design. Object-oriented techniques have been very successful in managing the complexity of modern software, but they have not resulted in significant amounts of cross-project code reuse. Sharing object-oriented code is difficult because of language incompatibilities, the lack of standardization for inter-object communication, and the need for compile-time coupling of interfaces. Component-based software development addresses issues of language independence—seamlessly combining components written in different programming languages—and component frameworks define standards for communication among components.

These advantages are especially appealing in high-performance scientific computing, where high-fidelity, multi-physics simulations are increasingly complex and often require the combined expertise of multidisciplinary research teams working in areas such as mathematical modeling, adaptive mesh manipulations, numerical linear and nonlinear algebra, optimization, load balancing, computational steering, parallel I/O, sensitivity analysis, visualization, and data analysis. Consequently, the interoperability and rapid application development afforded by component programming are of particular importance, as they help to support incremental shifts in parallel algorithms and programming paradigms that inevitably occur during the lifetimes of scientific application codes. In addition, since components can be configured to execute in remote locations, component programming can offer high-level abstractions that facilitate the use of distributed supercomputing resources, which have been shown to offer powerful potential [21].

Many differing opinions about component definitions exist within the software community [7, 47]. We present some working definitions as preliminaries for further discussion.

*This work has been partially supported by the MICS Division of the U.S. Department of Energy through the DOE2000 Initiative. For further information on the Common Component Architecture Forum, see `http://www.acl.lanl.gov/cca-forum` or write to `cca-forum@z.ca.sandia.gov`.

†Sandia National Laboratories, `rob@z.ca.sandia.gov`.

‡Indiana University, `gannon@cs.indiana.edu`

§Oak Ridge National Laboratory, `geist@msr.epm.ornl.gov`.

¶Advanced Computing Laboratory, Los Alamos National Laboratory, `kate@lanl.gov`.

‖Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, `skohn@llnl.gov`.

**Mathematics and Computer Science Division, Argonne National Laboratory, `mcinnes@mcs.anl.gov`.

††Department of Computer Science, University of Utah, `sparker@taz.cs.utah.edu`.

‡‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, `smolinski1@llnl.gov`.

- A *component* is an independent unit of software deployment. It satisfies a set of behavior rules and implements standard component interfaces that allow it to be composed with other components. These behavior rules are often specified as design patterns that must be followed when writing the component.

- A *component integration framework* is an implementation of a set of interfaces and rules of interaction that govern the communication among components.

- A *component architecture* is a specification of a set of interfaces and rules of interaction that govern the communication among components and other necessary tools, such as repositories and composition tools.

We have recently established the Common Component Architecture (CCA) Forum [15], a group whose current membership is drawn from various Department of Energy national laboratories and collaborating academic institutions. The goal of the CCA Forum is to specify a component architecture for high-performance computing, where our target architectures include workstation networks, distributed-memory multiprocessors, clusters of symmetric multiprocessors, and remote resources. We hope that this work will lay a foundation for the definition of standardized sets of domain-specific component interfaces and for the interoperability among toolkits developed by different teams across different institutions. The purpose of this paper is to discuss the current CCA specification and to present progress of the group to date.

The software industry has defined component standards such as CORBA [40], COM [45], and JavaBeans [19] to address similar complexities within their target applications (see Section 3 for a detailed discussion). Our approach leverages this work where appropriate, but addresses the distinctly different technical challenges of large-scale scientific simulations. Based on the lessons learned from research projects in high-performance component architectures by CCA participants (see, e.g., [3, 44, 25, 32, 36, 37]) and projects considering related design issues (see, e.g., [1, 23, 26, 6]), we are developing a single component interface specification that will enable interactions among scientific components that follow this standard. Additional related work [10, 8, 22, 35] can be found elsewhere.

We recognize two levels of interoperability: *component-level* interoperability, for which all the vital functions of any *one* architecture are accessible to any compliant component through a standard interface (e.g., facilities available within a CORBA ORB), and *framework-level* interoperability, for which the frameworks themselves interoperate through a standardized interface (e.g., inter-ORB communication via CORBA IIOP). Providing component-level interoperability requires defining an interaction model common to all components and a small set of indispensable high-level framework services. In addition to these requirements, framework-level interoperability necessitates the standardization of a number of low-level services. Since defining a standard for interoperability at the framework level requires a superset of features needed for the component level, our focus is on providing the latter now and extending it in the future to include framework-level interoperability features. The scope of this paper is limited to component-level interoperability.

The remainder of this paper motivates and explains our approach, beginning in Section 2 with a discussion of some of the challenges in large-scale scientific computing. Section 3 compares our strategy with related work in the software industry. Section 4 presents a high-level view of the CCA standard and provides a roadmap outlining the relationships among its constituents. Sections 5 and 6 describe in detail the parts of the CCA standard that are most crucial for defining component interactions in high-performance scientific software, namely, a scientific interface definition language and a "ports" component linking and composition model with direct-connect and collective capabilities. Finally, Section 7 outlines future directions of work.

2 Motivating Examples

Our work is motivated by collaborations with various computational science research teams, who are investigating areas such as combustion [14], microtomography [48], particle beam dynamics [30], mold filling [31], and plasma simulation [43]. In conjunction with theoretical and experimental research, these simulations are playing increasingly important roles in overall scientific advances, particularly in fields where experiments are prohibitively expensive, time consuming, or in some cases impossible. While each of these simulations requires different mathematical models, numerical methods, and data analysis techniques, they could all benefit from infrastructure that is more flexible and extensible and therefore better able to manage complexity and change.

To enable a more concrete discussion of the CCA approach, we briefly review some challenges arising in chemically reacting flow simulations, which have demanding requirements for high resolution and complex physical submodels for turbulence, chemistry, and multiphase flows. Section 2.1 presents current functionality of a particular application, while Section 2.2 describes potential enhancements that component-based technology could help to support.

2.1 Computational Hydrodynamics Example

We consider the CHAD (Computational Hydrodynamics for Advanced Design) application [14, 42] because it exhibits computational requirements common within many of high-performance scientific codes. CHAD has been developed for fluids simulations in the automotive industry under the Supercomputing Automotive Applications Partnership with the United States Council for Automotive Research and five Department of Energy national laboratories (Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, and Sandia). CHAD is the successor of KIVA [2], which has become a standard tool for device-level modeling of internal combustion engines. CHAD is intended for automotive design applications such as combustion, interior airflow, under-hood cooling, and exterior flows. Currently, CHAD solves the single-phase, compressible Navier-Stokes equations using an arbitrary Lagrangian-Eulerian formulation with hybrid unstructured meshes and a finite volume discretization scheme. The application was designed from its inception as parallel code using `Fortran 90` and encapsulation of nonlocal communication in gather/scatter routines using the Message Passing Interface (MPI) standard [39].

2.2 Component Challenges and Opportunities

CHAD researchers are experimenting with numerical strategies ranging from explicit through semi-implicit and even more fully implicit schemes using Newton-type methods. Using semi-implicit and implicit techniques helps to overcome stability and accuracy restrictions on computational timesteps, and thereby can often help to reduce overall time to solution.

Figure 1 demonstrates some typical interactions among components for a semi-implicit solution procedure within a PDE-based simulation. While a single diagram cannot express the richness of interactions within CHAD, nor the range of functionality needed by our motivating applications, this picture does convey key themes that motivate the CCA approach. We focus on (1) *fast interactions* between components via a "ports" component linking and composition model that allows direct connections (see Section 6.2), and (2) *collective interactions* among components that use multiple processes or threads (see Section 6.3). Collective abstractions are important for communication between both tightly coupled and loosely coupled components. For example, Figure 1 demonstrates collective directly connected ports between parallel preconditioner and Krylov solver components. The diagram also shows collective distributed port communication between numerical components of a parallel application and remote visualization tools.

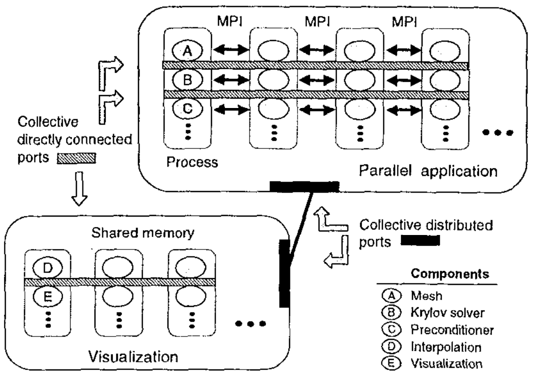

Figure 1. Diagram of component interactions. Parallel numerical components that use distributed data structures and require interconnections with low latency and high bandwidth are represented in the upper portion of the figure. Components for visualization, which can often be more loosely coupled and differently distributed than the numerical components, are shown in the figure's lower portion. Communication within a parallel component is at the discretion of the component itself. For example, in this diagram component A (a mesh) uses MPI to communicate among the four processes over which it is distributed, while component E (a visualization tool) uses shared memory. Communication between components is handled by ports.

The goals of the CCA Forum are to simplify the infusion of new techniques within the lifetimes of existing applications such as CHAD and to facilitate the construction of new models. Interactions among multiple tools that use current-generation infrastructure typically require labor-intensive translations between interfaces and data structures. We aim to simplify this process and also to enable dynamic interactions, since researchers may wish to introduce new components during the course of ongoing simulations. For example, a researcher may wish to visualize flow fields on a local workstation by dynamically attaching a visualization tool to an ongoing simulation that is running on a remote parallel machine. Upon observing that the flow fields are not converging as expected, the researcher may wish to introduce a new scheme for hierarchical mesh refinement.

One of the most computationally intensive phases within the semi-implicit and implicit strategies under consideration within CHAD is the solution of discretized linear systems of the form $Ax = b$, which are very large and have sparse coefficient matrices A. The Equation Solver Interface (ESI) Forum [20] is defining collections of abstract interfaces for solving such systems, with a goal of enabling applications like CHAD to experiment more easily with multiple solution strategies and to upgrade as new algorithms with better latency tolerance or more efficient cache utilization are discovered and encapsulated within toolkits. This area is one of many (e.g., partitioning, mesh management, discretization, optimization, visualization) that could benefit from component-based infrastructure to facilitate experiments among different tools.

3 Relationship to Existing Standards

Component architecture standards such as CORBA [40], COM [45], and JavaBeans [19] have been defined by industrial corporations and consortia and are employed by millions of users. Unfortunately, these standards do not address the needs of high-performance scientific computing, primarily because they do not support efficient parallel communication channels between components. Abstractions suitable for high-performance computing are needed. The existence of many successful high-performance languages and libraries—such as HPC++ [24], POOMA [4], ISIS++ [12], SAMRAI [29], and PETSc [5]—testifies that such abstractions can enable the user to develop more efficient programs faster. Similarly, we need abstractions that capture high-performance concepts in component architectures. For example, PARDIS [37] and PAWS [6] successfully show that introducing abstractions for single program multiple data (SPMD) computation can enable more efficient interactions between SPMD programs. In this section, we briefly review these industry standards and explain their limitations for high-performance scientific computing.

3.1 Microsoft COM and ActiveX

COM (Component Object Model) is Microsoft's component standard that forms the basis for interoperability among all Window-based applications. ActiveX [11] defines standard COM interfaces for compound documents. Microsoft has developed a distributed version of COM, called DCOM, that targets networked Windows workstations.

COM targets business objects and does not include abstractions for parallel data layout or basic scientific computing data types, such as complex numbers and Fortran-style dynamic multidimensional arrays. Also, COM does not easily support implementation inheritance and multiple inheritance (which can be implemented through aggregation or containment). Some scientific libraries (see, e.g., [20]) require multiple inheritance and a simple model for polymorphism, which COM does not provide.

3.2 Sun JavaBeans and Enterprise JavaBeans

JavaBeans and Enterprise JavaBeans (EJB) are component architectures developed by Sun and its partners. They are based on Sun's `Java` programming language and are cross-platform competitors to Microsoft's COM.

Neither JavaBeans nor EJB directly addresses the issue of language interoperability, and therefore neither is appropriate for the scientific computing environment. Both JavaBeans and EJB assume that all components are written in the `Java` language. Although the Java Native Interface [34] library supports interoperability with C and C++, using the Java virtual machine to mediate communication between components would incur an intolerable performance penalty on every intercomponent function call.

3.3 OMG CORBA

CORBA is a distributed object specification supported by the OMG (Object Management Group), a consortium of over eight hundred partners. CORBA supports the interaction of complex objects written in different languages distributed across a network of computers running different operating systems.

The current CORBA specification does not define a component model, although a CORBA 3.0 component specification [41] is currently under review by the OMG. Like COM, CORBA does not provide abstractions necessary for high-performance scientific computing, such as Fortran-style dynamic multi-dimensional arrays and complex numbers. Although CORBA enables robust and efficient implementations for distributed applications, it is far too inefficient when a method call is made within the same address space. While a recently established high-performance CORBA working group [28] may eventually address a subset of our performance concerns, their mandate does not address the range of parallel computing issues, as discussed in Section 2. CORBA also has a limited object model in that method overriding is not supported and the semantics of multiple implementation inheritance can lead to ambiguities.

While CORBA 2.0 does not provide for a component interaction mechanism, the CCA specification does. It should be observed that the CORBA object model is sufficiently powerful to suppport an implementation of the CCA. This is a good example of the intent of the CCA specification: a layer on top of an existing system that enables high-performance computing. Such a "CCA over CORBA" implementation, targeting distributed environments, is being planned by one of the participating forum members.

4 Overview of the CCA Standard

We define the Common Component Architecture as a set of specifications and their relationships as depicted in Figure 2. The elements with gray background pertain to specific implementations of a component architecture, while the elements with white background depict parts of the CCA standards necessary for component-level interoperability.

As shown in the picture, components interact with each other and with a specific framework implementation through standard application programming interfaces (APIs). Each component can define its inputs and outputs by using a *scientific interface definition language (SIDL)*; these definitions can be deposited in and retrieved from a repository by using a *CCA Repository API*. The repository API defines the functionality necessary to search a framework repository for components as well as to manipulate components within the repository. In addition, these component definitions can serve as input to a proxy generator that generates component stubs, which form the component-specific part of the *CCA Ports*. Components can use framework services directly through the *CCA Services* interface. The *CCA Configuration API* supports interaction between components and various builders for functions such as notifying components that they have been added to a scenario and deleted from it, redirecting interactions between components, or notifying a builder of a component failure.

A component framework is said to be CCA compliant if it conforms to these standards—that is, provides the required CCA services and implements the required CCA interfaces. Different components require different sets of services to interoperate. For example, some will require remote communication while others communicate only in the same address space. Therefore, the CCA standard will allow different flavors of compliance; each component will specify a minimum flavor of compliance required of a framework within which it can interact.

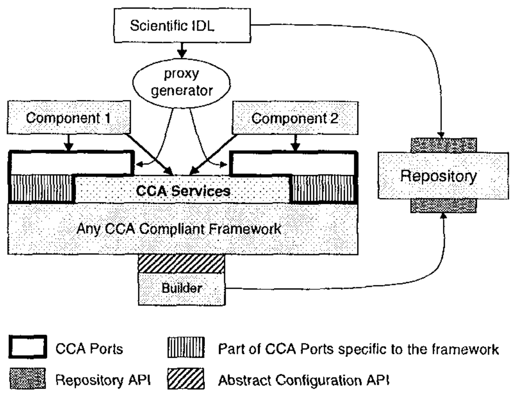

Figure 2. Relationships among CCAelements.

We will now describe in some detail three elements of the CCA standard that we believe are most critical for high-performance scientific computing, namely, a scientific interface definition language, a ports model, and a minimal set of supporting services. Work on the other parts of the CCA standard is also in progress, but details are beyond the scope of this paper.

- *SIDL* is a programming-language-neutral interface definition language used to describe component interfaces. The SIDL provides a method for describing component and framework interfaces that is independent of the underlying implementation programming languages. Component descriptions using SIDL can be used by repositories and by a proxy generator to provide the component stubs element of communication ports.

- *CCA Ports* define the communication model for all component interactions. Each component defines one or more ports to describe the calling interface. Communication links between components are implemented by connecting compatible ports, where port compatibility is defined as object-oriented type compatibility of the port interfaces, as can be described in the SIDL. As shown in Figure 2, each port has two parts. The first part is a set of framework-specific but component-independent functionality pertaining to component interaction (e.g., adding a listener to an object) and has the same API for every component. The second part implements component-specific but framework-independent functionality; this part can be generated automatically by a proxy generator based on the component definition expressed in SIDL, and is referred to as a component stub. For example, a component stub may contain marshaling functions in a distributed environment.

- *CCA Services* present a framework abstraction that can be used in the component stub implementation as well as by the components themselves; this CCA element provides a clear definition of the *minimal* services a framework must implement in order to be CCA compliant. Two critical concerns guiding this design are that the services enable high-performance interactions and that the services are sufficiently compact and user friendly to enable a rapid learning process for component writers, many of whom will not be computer scientists. As such, we have identified that the key CCA services are creation of CCA Ports and access to CCA Ports, which in turn enable connections between components.

Additional common facilities to handle naming, relationship management, error handling, querying, and so forth are of course also important, because in practice many components would need and could share these facilities. However, because the particular needs of different components and frameworks vary considerably depending on usage environment, discussion of these issues is beyond the scope of this paper.

The following sections describe these features in more detail. A reference implementation is tracking the evolution of the Common Component Architecture. Likewise, several ongoing computational science projects are experimenting with the CCA to manage interoperability among components developed by different research groups; these experiences will motivate further extensions and refinements to design.

5 The Scientific IDL

The Scientific Interface Definition Language is a high-level description language used to specify the calling interfaces of software components and framework APIs in the component architecture. SIDL provides language interoperability that hides language dependencies to simplify the interoperability of components written in different programming languages. With the proliferation of languages used for numerical simulation—such as `C`, `C++`, `Fortran 77`, `Fortran 90`, `Java`, and `Python`—the lack of seamless language interoperability can be a significant barrier to developing reusable scientific components.

For the purposes of our high-performance scientific component architecture, SIDL must be sufficiently expressive to represent the abstractions and data types common in scientific computing, such as dynamically dimensioned multidimensional arrays and complex numbers. Unfortunately, no such IDL currently exists, since most IDLs have been designed for operating systems [17, 18] or for distributed client-server computing in the business domain [33, 40, 46].

The basic design of our scientific IDL borrows many concepts from current standards, such as the CORBA IDL [40] and the `Java` programming language [27]. This approach allows us to leverage existing IDL technology and language mappings. For example, CORBA already defines language mappings to `C`, `C++`, and `Java`, and ILU [33] (which supports the CORBA IDL) defines language mappings to `Python`.

The scientific IDL provides additional capabilities necessary for scientific computing [13, 38]. It supports object-oriented semantics with an inheritance model similar to that of `Java` with multiple interface inheritance and single implementation inheritance. IDL support for multiple inheritance with method overriding is essential for scientific libraries that exploit polymorphism through multiple inheritance, such as used in the Equation Solver Interface [20] standard. The IDL and associated run-time system provide facilities for cross-language error reporting. We have also added IDL primitive data types for complex numbers and multidimensional arrays for expressibility and efficiency when mapping to implementation languages.

We are developing SIDL support for reflection and dynamic method invocation, which are important capabilities for a component architecture. Interface information for dynamically loaded components is often unavailable at compile time; thus, components and the associated composition tools and frameworks must discover, query, and execute methods at run time. The SIDL reflection and dynamic method invocation mechanisms are based on the design of the `Java` library classes in `java.lang` and `java.lang.reflect`. Reflection information for every interface and class will be generated automatically by the SIDL compiler based on IDL descriptions.

Our SIDL implementation currently supports language mappings for both `C` and `Fortran 77`, and support for `C++` is under development. The `Fortran 77` language mapping is similar to the `C` language mapping defined by CORBA except that SIDL interfaces and classes are mapped to `Fortran` integers instead of opaque data types. The SIDL run-time environment automatically manages the translation between the `Fortran` integer representation and the actual object reference. The `Fortran 90` language mapping is still under development. `Fortran 90` is a particular challenge for scientific language interoperability, because `Fortran 90` calling conventions and array descriptors vary widely from compiler to compiler.

6 Component Interaction through Ports

Every component architecture is characterized by the way in which components are composed together into applications. As introduced in Section 4, CCA Ports are communication end points that define the connection model for component interactions. Within Figure 1, ports define the interactions between relatively tightly coupled parallel numerical components, which typically require very fast communication for scalable performance; ports also define loosely coupled interactions with possibly remote components that monitor, analyze, and visualize data.

To address this range of requirements, we adopt a *provides/uses* interface exchange mechanism, similar to that within the CORBA 3.0 proposal [41]. This approach enables connections that do not impede inter-component performance, yet allows a framework to create distributed connections when desired. In the ideal case, an attached component would react as quickly as an inline function call. We refer to this situation as *direct connection*, which is further discussed in Section 6.2. This type of connection makes the most sense when the component instances exist in the same address space. Loosely coupled distributed connections should be available through the very same interface as the tightly coupled direct connections, without the components being aware of the connection type. This need arises because high-performance components will often be parallel programs themselves. A parallel component may reside inside a single multiprocessor or it may be distributed across many different hosts. Existing component models have no concept of attaching two parallel components together, and existing research systems, such as CUMULVS [26], PAWS [6], and PARDIS [37], approach this problem in different ways. We therefore introduce a *collective port* model to enable interoperability between parallel components, as discussed in Section 6.3.

In the JavaBeans model [19], components notify other listener components by generating events. Components that wish to be notified of events register themselves as listeners with the target components. Although there are some similarities to the CCA specification, JavaBeans does not allow a *provides/uses* design pattern as part of its standard. In the COM/DCOM model [45], one component calls the interface functions exported by another. The COM model is very similar in form to the CCA specification. Platform interoperability issues are, in the opinion of the CCA working group, important enough that COM has not been not adopted outright. In the proposed CORBA 3.0 component model [41], both events and a *provides/uses* interface model are used. The *provides/uses* pattern employed by the CCA is very close to this proposed approach, and any component that is CCA compliant will likely map easily to CORBA 3.0. However, at the time of this writing, CORBA 3.0 is a proposed standard that is still undergoing rapid change, and CORBA 3.0 may see no implementation for years. The CCA working group believes that a compatible standard for high-performance computing should appear much more quickly than the CORBA 3.0 time frame. For this reason we have chosen the *provides/uses* pattern for use as the CCA Ports architecture. It is expected (and hoped) that the CORBA 3.0 specification will not drift far from what is described here.

6.1 The Basics of CCA Ports

The concept of CCA Ports arises from the data-flow world, where component interactions are limited to pipelining data from one component to the next. CCA Ports generalize this idea to admit method calls and return values along the pipeline, allowing for a richer variety of component interactions. Links between components are implemented by a *provides/uses* interface design pattern, which is flexible enough to allow direct component interface connections for high performance or connections through proxy intermediaries enabling distributed object interactions. Significantly, in the CCA model, port connection is the responsibility of the framework; therefore, a particular component may find itself connected in a variety of different ways depending on its environment and mode of use (see [9] for details of the CCA ports specification and an applet demonstration).

In the CCA architecture, components are linked together by connecting a "port" interface from one component to a "port" interface on another. As demonstrated in Figure 3, we employ two types of ports:

- *Provides* port. A *Provides* port is an interface that a component provides to others.
- *Uses* port. A *Uses* port interface has methods that one component (the caller) wants to call on another component (the callee); the caller component retrieves the *Uses* interface from the CCA Services handle.

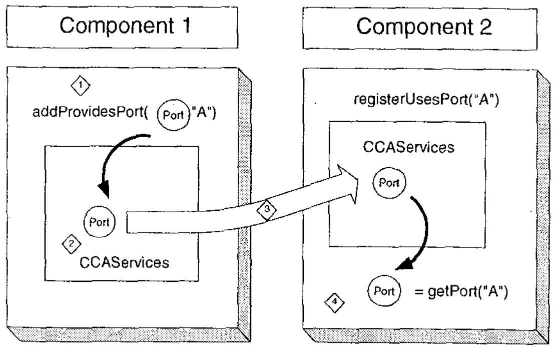

Figure 3. Illustration of the connection mechanism. ⟨1⟩ The provided interface (i.e., `ProvidesPort`) is made known to Component 1's containing framework by ⟨2⟩ passing it to the `CCAServices` handle via the `addProvidesPort()` method. ⟨3⟩ At the framework's option, either the interface or a proxy for the interface can be given to Component 2 through its `CCAServices` handle. ⟨4⟩ Component 2 retrieves the interface using the `getPort()` method.

Provides ports are generalized listeners in the sense that they listen to *Uses* interfaces (i.e., calls of their functions by another component). Each *Uses* port maintains a list of listeners. To connect one component to another, one adds a *Provides* (input) port of one component to another's *Uses* (output) port. This approach follows many features of the proposed CORBA 3.0 design. When a component calls a member function on one of its *Uses* ports, the same member function on each listening *Provides* port is called. Note that this means one call may correspond to zero or more invocations on provider components.

As introduced in Section 4, all interaction between the component and its containing framework will occur through the component's `CCAServices` object, which is set by the containing framework. The component creates and adds *Provides* ports to the `CCAServices`, and registers and retrieves *Uses* ports from the `CCAServices`. The `CCAServices` enables access to the list of *Provides* and *Uses* ports and to an individual port by its instance name. It also implements a method for obtaining the various ports and registering them with the framework.

6.2 Direct-Connect Ports

Much of the reason for adopting the *provides/uses* interface exchange mechanism for connecting CCA components is to enable high-performance computing. Except for the SIDL bindings to `UsesPort` and `ProvidesPort` interfaces, the overhead for the privilege of becoming a CCA component is nothing more than a direct function call to the connected object. That is, there is no penalty for using the *provides/uses* component connection mechanism proposed in the CCA specification. The cost of the intervening SIDL binding for language independence is estimated to be approximately 2-3 function calls per interface method call.

Components can be directly connected in a variety of ways; probably the simplest is to create an object that exports a `DirectConnectPort` interface subclassing both the `UsesPort` and `ProvidesPort` interfaces. This way the framework gets a *Provides* interface from one component and gives that same interface directly to a connecting component as a *Uses* interface. Note that with this approach the framework still retains full control over the connection between components. Optionally, the provided `DirectConnectPort` can be translated through a proxy by a separate `UsesPort` provided by the framework, without the components on either end of the connection needing to know.

6.3 Collective Ports

The concept of *Collective Ports* is a small but powerful extension of the basic CCA Ports model to handle interactions among parallel components and thereby to free programmers from focusing on the often intricate implementation-level details of parallel computations. The *provides/uses* port interfaces and other port information are accessible from every thread or process in a parallel component. The CCA standard does not place any restrictions on the means by which particular implementations address this. For example, in a distributed-memory model a copy of these classes could be maintained by every process participating in computation, whereas in shared memory a class could be represented just once. However, the CCA standard does require that as one of the CCA services the implementation maintain consistency among the classes.

The creation of a collective port requires that the programmer specify the mapping of data (or processes participating) in the operations on this port. In the most common case the mappings of the input and output ports match each other. For example, n processes or threads in one component are mapped to n processes or threads in the other, and in this case data would not need redistribution between the parallel components. In the second most common case, a serial component interacts with a parallel component. The semantics of this interaction are very similar to broadcast, gather, and scatter semantics used in collective communication. Collective ports are defined generally enough to allow data to be distributed arbitrarily in the connected components; as demonstrated in Figure 1, this capability is useful in connecting a parallel numerical simulation with differently distributed visualization tools. We are investigating issues in the behavior of information flow between collective ports, especially in cases of mismatch in cardinality, time, and space.

7 Future Directions

This discussion has introduced the foundation for research by the CCA forum in defining a common component architecture that supports the needs of the high-performance scientific computing community and leverages existing component standards, but will likely not be addressed by them. Key facets of this work are development of an IDL that supports scientific abstractions for component interface specification and definition of a ports connection model that supports collective interactions. This architecture enables connections that do not impede inter-component performance, yet allows a framework to create distributed connections when desired. Currently, we are implementing various Ports subclasses that relate directly to high-performance computing. Among these are the collective ports discussed earlier, a component based on a numeric solvers standard [20], and a reference implementation of a CCA-compliant framework (see [15] for further information). Other proposals for components and standard interfaces compliant with the current CCA Ports specification are openly solicited.

Future plans include incorporating support for different computational models (e.g., SPMD and threaded models) and extending the definition of CCA Ports to accommodate dynamic component hook-up and configuration. Some changes to the existing port specification are inevitable as we gain experience with actual high-performance components. Currently, the CCA specification makes no provision for framework services beyond Ports. At this moment a proposal is being crafted for gaining access through the existing CCA specification to services provided by existing frameworks, such as CORBA or Enterprise JavaBeans. It does not seem likely that the CCA working group will decide to *require* any of these services to be present. This is because high-performance environments are often exotic, and requiring services may limit some of the intended audience for this specification.

Beyond these modifications and clarifications to the existing standard, the CCA working group will function as a standards body, incorporating or rejecting proposed port and component additions to the essential core of the standard. This phase of our activity has just begun, but is vital to the success of our mission. Our goal is to incorporate enough standard interfaces and components to make plug-and-play high-performance computing a reality. This is an impossibly tall order for the CCA members to accomplish by themselves. However, by incorporating components and interfaces from interested researchers and consortia, it is hoped that this vision can be realized.

Acknowledgments

The Common Component Architecture (CCA) Forum was initially inspired by the DOE2000 Initiative [16] and is motivated by ongoing collaborations with various scientific research groups. We especially thank Tom Canfield for conveying some of the challenges in device-scale combustion modeling, as discussed in Section 2.

The CCA Forum comprises researchers from national laboratories within the Department of Energy and collaborating academic institutions; current participants are Argonne National Laboratory, Indiana University, Lawrence Berkeley National Laboratory, Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, Sandia National Laboratories, and the University of Utah. The ideas presented here were developed with the participation of various individuals at these institutions, including Ben Allan, Rob Armstrong, Pete Beckman, Randall Bramley, Robert Clay, Andrew Cleary, Dennis Gannon, Al Geist, Paul Hovland, Bill Humphrey, Kate Keahey, Jim Kohl, Scott Kohn, Lois McInnes, Bill Mason, Carl Melius, Brent Milne, Noël Nachtigal, Steve Parker, Mary Pietrowicz, Barry Smith, Steve Smith, Brent Smolinski, Brian Toonen, and John Wu. These ideas have been influenced by laboratory and university software development teams, some of whose members are represented above.

References

[1] ALICE Web page. `http://www.mcs.anl.gov/-alice`, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] A. A. Amsden, P. J. O'Rourke, and T. D. Butler. KIVA-II: A computer program for chemically reactive flows with sprays. Technical Report LA-11560-MS, Los Alamos National Laboratory, May 1989.

[3] R. C. Armstrong and A. Chung. POET (parallel object-oriented environment and toolkit) and frameworks for scientific distributed computing. In *Hawaii International Conf. on System Sci.*, 1997.

[4] S. Atlas, S. Banerjee, J. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high-performance distributed simulation environment for scientific applications. In *Supercomputing '95 Proceedings*, December 1995.

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[6] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*, July 1998.

[7] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component? *Software – Concepts and Tools*, 19:49–56, 1998.

[8] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. Application specific tools. chapter 7, in *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.

[9] CCA Ports Web page. `http://z.ca.sandia.gov/-~cca-forum/port-spec`.

[10] K. M. Chandy, A. Rifkin, P. A. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A world-wide distributed system using Java and the internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[11] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1997.

[12] R. L. Clay, K. Mish, and A. B. Williams. ISIS++ Web page. `http://ca.sandia.gov/isis`.

[13] A. Cleary, S. Kohn, S. Smith, and B. Smolinski. Language interoperability mechanisms for high-performance scientific computing. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998.

[14] J. P. Collins, P. Colella, and H. M. Glaz. Implicit-explicit Eulerian Godunov scheme for compressible flows. *J. Comp. Phys.*, 116:195–211, 1995.

[15] Common Component Architecture Forum. See `http://www.acl.lanl.gov/cca-forum`.

[16] DOE2000 Initiative. See `http://www.mcs.anl.gov/DOE2000`.

[17] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[18] E. Eide, J. Lepreau, and J. L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[19] R. Englander. *Developing Java Beans*. O'Reilly, June 1997.

[20] Equation Solver Interface Forum. See http://-z.ca.sandia.gov/esi/.

[21] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[22] G. Fox and W. Furmanski. Object-based approaches. chapter 10, in *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.

[23] L. A. Freitag, W. D. Gropp, P. D. Hovland, L. C. McInnes, and B. F. Smith. Infrastructure and interfaces for large-scale numerical software. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*. to appear (also available as Argonne preprint ANL/MCS-P751-0599).

[24] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. HPC++ and the HPC++Lib Toolkit. *Languages, Compilation Techniques and Run Time Systems (Recent Advances and Future Perspectives)*, to appear.

[25] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component architectures for distributed scientific problem solving. *IEEE Computational Science and Engineering*, 5(2):50–63, 1998.

[26] A. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing Fault Tolerance, Visualization and Steering of Parallel Applications. *The International Journal of Supercomputer Applications and High Performance Computing*, (11):224–235, 1997.

[27] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, 1996. Available at http://java.sun.com.

[28] High-performance CORBA Working Group. See http://www.omg.org/homepages/realtime/working_groups/high_performance_corba.html.

[29] R. Hornung and S. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998. See http://www.llnl.gov/CASC/SAMRAI.

[30] W. Humphrey, R. Ryne, J. Cummings, T. Cleland, S. Habib, G. Mark, and J. Qiang. Particle beam dynamics simulations using the POOMA framework. In *Proceedings of the ISCOPE '98 Conference*, 1998.

[31] F. Illinca, J.-F. Hetu, and R. Bramley. Simulation of 3-D mold-filling and solidification processes on distributed memory parallel architectures. In *Proceedings of International Mechanical Engineering Congress & Exposition*, 1997.

[32] InDEPS Web page. http://z.ca.sandia.gov/-~indeps/, Sandia National Laboratories.

[33] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi. *ILU Reference Manual*. Xerox Corporation, Nov. 1997. Available at ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[34] JavaSoft. *Java Native Interface Specification*, May 1997.

[35] A. Joshi, T. Drashansky, J. R. Rice, S. Weerawarana, and E. Houstis. Multiagent simulation of complex heterogeneous models in scientific computing. *Math. Comput. Simul.*, 44:43–59, 1997.

[36] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component architecture for high-performance applications. *International Journal of High-Performance and Scientific Applications*, to appear.

[37] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, pages 31–39, August 1997.

[38] S. Kohn and B. Smolinski. Component interoperability architecture: A proposal to the common component architecture forum. In preparation, 1999.

[39] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.

[40] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0.* OMG Document, June 1995.

[41] OMG. *Corba Components. Revision 3.0.* OMG TC Document orbos/99-02-05, March 1999.

[42] P. J. O'Rourke and M. S. Sahota. A variable explicit/implicit numerical method for calculating advection on unstructured meshes. *J. Comp. Phys.*, 143:312–345, 1998.

[43] W. Park, E. V. Belova, G. Y. Fu, X. Z. Tang, H. R. Strauss, and L. E. Sugiyama. Plasma simulation studies using multilevel physics models. *Physics of Plasmas*, 6:1796–1803, 1999.

[44] S. Parker, D. Weinstein, and C. Johnson. The SCIRun computational steering software system. In E. Arge, A. Bruaset, and H. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–44. Birkhauser Press, 1997.

[45] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

[46] J. Shirley, W. Hu, and D. Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[47] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1998.

[48] G. von Laszewski, M.-H. Su, J. A. Insley, I. Foster, J. Bresnahan, C. Kesselman, M. Thiebaux, M. L. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

Language Interoperability for High-Performance Parallel Scientific Components*

Brent Smolinski, Scott Kohn, Noah Elliott, and Nathan Dykman

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract. Component technologies offer a promising approach for managing the increasing complexity and interdisciplinary nature of high-performance scientific applications. Language interoperability provides the flexibility required by component architectures. In this paper, we present an approach to language interoperability for high-performance parallel components. Based on Interface Definition Language (IDL) techniques, we have developed a Scientific IDL (SIDL) that focuses on the abstractions and performance requirements of the scientific domain. We have developed a SIDL compiler and the associated run-time support for reference counting, reflection, object management, and basic exception handling. The SIDL approach has been validated for a scientific linear solver library. Initial timing results indicate that the performance overhead is minimal (less than 1%), whereas the savings in development time for interoperable software libraries can be substantial.

1 Introduction

The scientific computing community is beginning to adopt component technologies and associated programming methodologies [1, 2, 10, 17] to manage the complexity of scientific code and facilitate code sharing and reuse. Components require language interoperability to isolate component implementation details from applications. This ensures that applications and components can be created and evolve separately. With the proliferation of languages used for numerical simulation—such as C, C++, Fortran 90, Fortran 77, Java, and Python—the lack of seamless language interoperability negatively impacts the reusability of scientific codes.

Providing interoperability among the many languages used in scientific computing is a difficult problem for both component and library developers. Without language interoperability, application developers must use only the same language as the components, even though better languages may exist. If language

* Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. This work has been funded by LDRD grant 99-ERD-078. Available as LLNL techinical report UCRL-JC-134260.

interoperability is desired, component developers and users are often forced to write "glue code" that mediates data representations and calling mechanisms between languages. However, this approach is labor-intensive and in many cases does not provide seamless language integration across the various calling languages. Both approaches couple the components and applications too tightly, restricting component reuse and flexibility.

1.1 Language Interoperability Design Considerations

The design considerations associated with language interoperability for high-performance scientific computing differ from those of the business sector, which is supported by industry efforts such as COM [7, 15] and CORBA [16]. The Common Component Architecture (CCA) [1], Equation Solver Interface [9] and other scientific computing working groups require support for complex numbers, `Fortran`-style dynamic multidimensional arrays, object-oriented semantics with multiple inheritance and method overriding, and very efficient function invocation for components living in the same address space. The CCA consortium is developing component technologies appropriate for high-performance parallel scientific computing. The ESI is developing standards for linear solvers and associated preconditioners based on component approaches to increase the interoperability of numerical software developed by different development teams.

1.2 Related Interoperability Approaches

Several language interoperability packages have been developed that automatically generate glue code to support calls among a small set of targeted languages. For example, the SWIG package [3] reads `C` and `C++` header files and generates the mediating code that allows these routines to be called from scripting languages such as `Python`. Such approaches typically introduce an asymmetric relationship between the scripting language and the compiled language. Calls from the scripting language to the compiled language are straight-forward, but calls from the compiled language to the scripting language are difficult or are not supported.

Foreign invocation libraries have been used to manage interoperability among targeted languages. For instance, the *Java Native Interface* [13] defines a set of library routines that enables `Java` code to interoperate with libraries written in `C` and `C++`.

Such interoperability approaches support language interoperability among only a limited set of languages, and they do not support a single, universal mechanism that works with all languages. In the worst case, interoperability among N languages could require $O(N^2)$ different approaches. Component architectures require a more general approach, which we describe in the following section.

1.3 Interoperability Through an IDL Approach

One interoperability mechanism used successfully by the distributed systems and components community [12, 15, 16, 18] is based on the concept of an Interface Definition Language or IDL. The IDL is a new "language" that describes the calling interfaces to software packages written in standard programming languages such as C, Fortran, or Java. Given an IDL description of the interface, IDL compilers automatically generate the glue code necessary to call that software component from other programming languages. The advantage of an IDL approach over other approaches is that it provides a single, uniform mechanism for interoperability among a variety of languages.

Current IDL implementations are not sufficient for specifying interfaces to high-performance scientific components. First, standard IDLs such as those defined by CORBA and COM are targeted towards business objects and do not include basic scientific computing data types such as complex numbers or dynamic multidimensional arrays. Second, approaches focused on distributed objects do not generally provide support for high-performance, same address space function calls between different languages. Our performance goal is to reduce the overhead of single address space function calls to about that of a C++ virtual function invocation. Third, many IDLs do not support multiple inheritance or have a limited object model. For example, COM does not support multiple inheritance and supports implementation inheritance only through composition or aggregation, which can be computationally expensive and difficult to implement. CORBA does not support method overriding, which is required for polymorphism.

We have adopted an IDL approach for handling language interoperability in a scientific computing environment. We have developed a Scientific IDL called SIDL [6, 14] as well as a run-time environment that implements bindings to SIDL and provides the library support necessary for a scientific component architecture. Currently SIDL supports bindings to C and Fortran 77, although others are under development. Preliminary experiments with a scientific solver library have shown that SIDL is expressive enough for scientific computing and that language interoperability is possible with little measurable run-time overheads.

1.4 Paper Organization

This paper is organized as follows. Section 2 introduces SIDL features that are necessary for high-performance parallel computing. Section 3 describes the bindings of SIDL to C and Fortran 77, as well as the run-time environment, which includes a SIDL compiler and library support. Section 4 details the process of applying the SIDL interoperability approach to a scientific software library and provides parallel performance results for both C and Fortran. Finally, we conclude in Section 5 with an analysis of the lessons learned and the identification of future research issues.

2 Scientific Interface Definition Language

A scientific IDL must be sufficiently expressive to represent the abstractions and data types common in scientific computing, such as dynamic multidimensional arrays and complex numbers. Polymorphism—required by some advanced numerical libraries [9]—requires an IDL with an object model that supports multiple inheritance and method overriding. The IDL should also provide robust and efficient cross-language error handling mechanisms.

Unfortunately, no current IDLs support all these capabilities. Most IDLs have been designed for operating systems [7, 8] or for distributed client-server computing in the business domain [12, 16, 18] and not for scientific computing.

The design of our Scientific IDL borrows many ideas from the CORBA IDL [16] and the `Java` programming language [11]. `SIDL` supports an object model similar to `Java` with separate interfaces and classes, scientific data types such as multidimensional arrays, and an error handling mechanism similar to `Java` and CORBA. `SIDL` provides reflection capabilities that are similar to `Java`.

The following sections describe `SIDL` in more detail. An example of `SIDL` for a scientific preconditioning solver library is given in Figure 3 of Section 4.

2.1 Scientific Data Types

In addition to standard data types such as *int*, *char*, *bool*, *string*, and *double*, `SIDL` supports *dcomplex*, *fcomplex*, and *array*. An *fcomplex* is a complex number of type float, and a *dcomplex* is a complex number of type double. A `SIDL` *array* is a multidimensional array contiguous in memory, similar to the `Fortran`-style arrays commonly used in scientific computing. The *array* type has both a type, such as *int* or *double*, and a dimension, currently between one through four, inclusive. In comparison, CORBA supports only statically-sized multidimensional arrays and single-dimension sequences, and COM supports only pointer-based, ragged multidimensional arrays.

2.2 SIDL Object Model

The `SIDL` object model is similar to that of the `Java` programming language. We chose the `Java` object model for `SIDL` because it provides a simple model for multiple inheritance. `SIDL` supports both interfaces and classes. A `SIDL` class may inherit multiple interfaces but only one class implementation. This approach solves the ambiguity problems associated with multiple implementation inheritance in languages such as `C++`.

SIDL provides a new set of interface method declarations. These declarations provide optimization opportunities and increase the expressiveness of the IDL. Like `Java`, class methods may be declared **abstract**, **final**, or **static**. An **abstract** method is purely declarative and provides no implementation; an implementation must be provided by a child class. A **final** method is one that cannot be overridden by child classes. The **final** construct enables optimizations in the run-time system that eliminate potential dereferences to an overriding method. As in C++ or Java, **static** methods are associated with a class, not a class instance, and therefore may be invoked without an object. The **static** construct simplifies developing SIDL interfaces to legacy libraries that were written without object-oriented semantics.

2.3 Scoping and Exception Handling

Every class and interface belongs to a particular package scope. Packages in SIDL are similar to namespaces in C++ and packages in Java. The package construct is used to create nested SIDL namespaces. Packages help prevent global naming collisions of classes and interfaces that are developed by different code teams.

Component architectures require robust error handling mechanisms that operate across language barriers. We have designed an error reporting mechanism similar to Java. All exceptions in SIDL are objects that inherit from a particular library interface called *Throwable.* Error objects support more complex error reporting than what is possible with simple integer error return codes. Error conditions are indicated through an environment variable that is similar to CORBA.

2.4 Reflection

Reflection is the mechanism through which a description of object methods and method arguments can be determined at run-time. Reflection is an critical capability for component architectures, as it allows applications to discover, query, and execute methods at run-time. This allows applications to create and use components based on run-time information, and to view interface information for dynamically loaded components that is often unavailable at compile-time.

The SIDL run-time library will support a reflection mechanism that is based on the design of the Java library classes in java.lang and java.lang.reflect. The SIDL compiler automatically generates reflection information for every interface and class based on its IDL description. The run-time library will support queries on classes and interfaces that allow methods to be discovered and invoked at run-time.

3 Bindings and Implementation

SIDL defines component interfaces in a language-independent manner. For each programming language, we must define language mappings that map constructs in SIDL onto that target language. In this section, we describe the mappings of SIDL to C and Fortran 77, as well as the required library support for the run-time environment. We discuss only the more challenging aspects of the mappings and implementation; a complete specification can be found elsewhere [14].

3.1 Mappings to C and Fortran 77

Because SIDL is based on CORBA IDL, we were able to use the CORBA specification [16] as a guide in mapping many of the SIDL constructs into C. Fortran 77 mappings closely followed the C mappings, whith exceptions as described below. The mappings for complex numbers and multidimensional arrays to C and Fortran 77, which are not part of the CORBA IDL, where relatively straightforward.

Mapping SIDL classes and interfaces in C and Fortran 77 presented some interesting challenges, since neither language supports object-oriented features. However, the IDL approach allows object-oriented concepts to be mapped onto non-object-oriented languages. For C, SIDL classes and interfaces are mapped to opaque structure pointers that encapsulate private data members, method invocation tables, and other implementation details. For Fortran 77, classes and interfaces are mapped to integers that are used as handles. The run-time environment manages object information and automatically translates between the Fortran integer representation and the actual object reference. Methods on SIDL objects are invoked using a standard C or Fortran 77 function call with the object reference as the first parameter. Figure 3 of Section 4 illustrates these conventions for a scientific linear solver library.

3.2 Implementing the SIDL Run-Time Environment

Much of the effort in developing the SIDL compiler and run-time system was in implementing the object model, namely: virtual function tables, object lookup table for mapping to and from Fortran integer handles, reference counting, dynamic type casting, exception handling mechanism, and reflection capabilities. The run-time library support is implemented in C and the compiler is written in Java. The "glue" code generated from the compiler is in C.

All object support is distributed between the glue code and the run-time library. The glue code contains the implementation of the object mapping, including the virtual function lookup table (similar to a C++ virtual function table), constructors, destructors, and support for dynamic type casting. The run-time library contains support for reference counting, object lookup mechanisms necessary for Fortran objects, and exception handling mechanisms. The reflection capability is supported through both the glue code and the run-time library.

One of the goals of the SIDL run-time environment is to provide extremely fast function calls between components living in the same memory space. For C to C calls, our current implementation requires one table look-up (to support virtual functions) and one additional function call. Calls between C and another language add the overhead of an additional function call, and calls between two non-C languages requires yet another call. These additional function calls are needed to isolate language-specific linker names. Where possible, the SIDL compiler takes advantage of the **static** and **final** qualifiers in SIDL by eliminating a function table lookup to functions for those types.

4 Applying SIDL to a Scientific Library

As a test case, we used the SIDL tools to create new interfaces for a semicoursening mulitigrid (SMG) solver [4], a preconditioner that is part of the *hypre* linear solver library [5]. *hypre* is a library of parallel solvers for large, sparse linear systems being developed at Lawrence Livermore National Laboratory's Center for Applied Scientific Computing. The library currently consists of over 30,000 lines of C code, and it has 94 encapsulated user-interface functions. To test our approach, we created a SIDL interface and and created both C and Fortran 77library wrappers with SIDLWe ran similar test drivers for the two SIDL generated wrappers and the original C interface already provided by the library, and compared the results from all three runs.

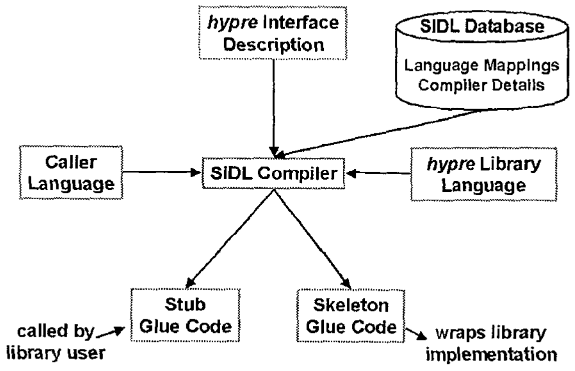

Fig. 1. Generating "glue" code for the *hypre* library using the SIDL tools.

Wrapping *hypre* using SIDL proceeded in three steps. First, the existing *hypre* interface was written in SIDL by two people, one who was familiar with SIDL and another who was familiar with the *hypre* library. The second step was to run the SIDL compiler with the interface description as input to automatically generate the glue code for each class (see Figure 1). Since the names created by SIDL compiler are slightly different from those expected by the rest of the original *hypre* library, the library had to be slightly modified to match the new names of the SMG interfaces. This step is not required if SIDL conventions are used and only has to be done once. Once the function calls were manually added for the C language bindings, the Fortran interface was created automatically by running the compiler once more with options for Fortran. The final step was to compile and link the drivers with the skeletons, stubs, and the *hypre* library.

We rewrote an existing SMG test driver to test the performance of the new interfaces. The driver uses SMG to solve Laplace's equation on a 3-D rectangular domain with a 7-point stencil. First, all calls in the existing C driver to the *hypre* library were replaced with the new C interfaces created by SIDL. Then we wrote a new Fortran driver for the same problem that calls the same *hypre* functions

```
package hypre {
   class stencil {
      stencil NewStencil(in int dim, in int size);
      int SetStencilElement(in int index, inout array<int> offset);
   };
   class grid {
      grid NewGrid(in mpi_com com, in int dimension);
      int SetGridExtents(inout array<int> lower, inout array<int> upper);
   };
   class vector {
      vector NewVector(in mpi_com com, in grid g, in stencil s);
      int SetVectorBoxValues(inout array<int> lower,
         inout array<int> upper, inout array<double> values);
      ...
   };
   class matrix { /* matrix member functions omitted in this figure */ };
   class smg_solver {
      int Setup(inout matrix A, inout vector b, inout vector x);
      int Solve(inout matrix A, inout vector b, inout vector x);
      ...
   };
};
```

Fig. 2. Portions of the *hypre* interface specification written in SIDL.

via the new Fortran interface. Figure 2 shows a portion of the *hypre* interface written in SIDL, and Figure 3 shows portions of both the C and Fortran drivers that call the *hypre* library using the automatically generated interfaces.

Both test drivers produced the same numerical results. We compared the efficiency of the new C and Fortran drivers to the original C driver. The drivers that used SIDL solved large problems—both sequentially and in parallel on 216 processors—with no noticeable effect (less than 1%) on the speed of execution. The overhead added by SIDL is negligible when compared to the overhead of the numerical kernels in the library.

This entire process required less than an afternoon to generate the SIDL interface, edit the skeleton code, and generate C and Fortran stub code. To put this in perspective, there was an effort by the *hypre* team to manually generate a Fortran interface for *hypre* that required over one person-week of effort. This work was targeted at the Solaris platform. Porting this hand-generated Fortran interface to another platform required a substantial re-write of the interface due to differences in Fortran name representation. Such platform dependencies are managed automatically by the SIDL tools.

C Test Code

```
hypre_vector b, x;
hypre_matrix A;
hypre_smg_solver solver;
hypre_stencil s;

b = hypre_vector_NewVector(com, grid, s);
...
x = hypre_vector_NewVector(com, grid, s);
...
A = hypre_matrix_NewMatrix(com, grid, s);
...

solver = hypre_smg_solver_new();
hypre_smg_solver_SetMaxItr(solver, 10);
hypre_smg_solver_Solve(solver, &A, &b, &x);
hypre_smg_solver_Finalize(solver);
```

Fortran 77 Test Code

```
integer b, x
integer A
integer solver
integer s

b = hypre_vector_NewVector(com, grid, s)
...
x = hypre_vector_NewVector(com, grid, s)
...
A = hypre_matrix_NewMatrix(com, grid, s)
...

solver = hypre_smg_solver_new()
hypre_smg_solver_SetMaxItr(solver, 10)
hypre_smg_solver_Solve(solver, A, b, x)
hypre_smg_solver_Finalize(solver)
```

Fig. 3. Sample test code calling *hypre* interfaces for C and Fortran 77 generated automatically using the SIDL tools.

5 Lessons Learned and Future Work

We have presented SIDL, a scientific interface definition language, and a run-time that meets the requirements requirements for scientific computing. SIDL borrows heavily from the CORBA IDL and Java programming language, while adding features necessary for scientific computing. SIDL seems to capture the abstractions necessary for scientific computing, as well as new features that a run-time can use to perform optimizations, which are not present in current IDL standards.

The SIDL run-time also provides fast same address space calls, which is important for effective scientific computation. A comparison using the *hypre* library showed that SIDL added only one to two percent overhead compared to the native interfaces. This is neglible when compared to the great savings in developer costs and flexibility. The SIDL run-time allowed the creation of a Fortran 77 interface in the *hypre* library in a fifth of the time required to create a similar interface by hand.

In the future we will develop bindings for C++, Java, Fortran 90, and Python and implement those bindings. Fortran 90 is challenging since Fortran 90 calling conventions vary widely from compiler to compiler. We will also continue our collaboration efforts with the CCA and ESI working groups. Other ESI specifications will require more expressability from SIDL than the *hypre* interface requires. Features may also need to be added to SIDL to support the specification of high-performance scientific components (e.g. CCA compliant components).

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, *Toward a common component architecture for high performance*

scientific computing, 1999.

2. S. Balay, B. Gropp, L. C. McInnes, and B. Smith, *A microkernel design for component-based numerical software systems*, in Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998.
3. D. M. Beazley and P. S. Lomdahl, *Building flexible large-scale scientific computing applications with scripting languages*, in The 8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997.
4. P. Brown, R. Falgout, and J. Jones, *Semicoarsening multigrid on distributed memory machines*, in SIAM Journal on Scientific Computing special issue on the Fifth Copper Mountain Conference on Iterative Methods, 1999.
5. E. Chow, A. Cleary, and R. Falgout, *Design of the hypre preconditioner library*, in Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998.
6. A. Cleary, S. Kohn, S. Smith, and B. Smolinski, *Language interoperability mechanisms for high-performance applications*, in Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998.
7. G. Eddon and H. Eddon, *Inside Distributed COM*, Microsoft Press, Redmond, WA, 1998.
8. E. Eide, J. Lepreau, and J. L. Simister, *Flexible and optimized IDL compilation for distributed applications*, in Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, 1998.
9. Equations Solver Interface Forum. See `http://z.ca.sandia.gov/esi/`.
10. D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju, *Component architectures for distributed scientific problem solving*, (1998).
11. J. Gosling and K. Arnold, *The Java Programming Language*, Addison-Wesley Publishing Company, Inc., Menlo Park, CA, 1996.
12. B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi, *ILU Reference Manual*, Xerox Corporation, November 1997. See `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.
13. JAVASOFT, *Java Native Interface Specification*, May 1997.
14. S. Kohn and B. Smolinski, *Component interoperability architecture: A proposal to the common component architecture forum. in preperation*, 1999.
15. MICROSOFT CORPORATION, *Component Object Model Specification (Version 0.9)*, October 1995. See `http://www.microsoft.com/oledev/olecom/title.html`.
16. OBJECT MANAGEMENT GROUP, *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.
17. S. Parker, D. Beazley, and C. Johnson, *The SCIRun Computational Steering Software System*, E. Arge, A.M. Bruaset, and H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997.
18. J. Shirley, W. Hu, and D. Magid, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

This article was processed using the LaTeX macro package with LLNCS style